FPGA Implementation of a 2D Strategy Game

Mehmet Kurt (2443497), Emirhan Yılmaz Güney (2443208), Tanel Gülerman (2443141)

***Abstract*—This paper provides our implementation of a 2-D Strategy Game in FPGA by using Verilog HDL. In the development stage, we firstly created a game logic by using state machine method and then implemented it in our game controller module. After completing the game logic part, we created the interface between the game outputs and VGA screen for visualization of the board and the game statistics. In these processes, we used Verilog HDL documentations, VGA working principle papers, and the pin assignment sheets of Altera DE-1 SoC board which we used for the implementation of the game. In addition to these major concepts, we also created a debouncer module and VGA sync module since it is required to drive the VGA screen and implement the game logic successfully.**

***Index Terms*—verilog, tictactoe, fpga, vga, strategy game, debouncing**

I. Introduction

IN this 314 Digital Electronics Laboratory Project, we have implemented the infamous game, "Triangles vs. Cricles". We have used Verilog HDL as description language and Altera DE-1 Soc Board as hardware. This project requires us to implement this game as an interactive turn based strategy game which has various built in condition checker codes in order to make sure gameplay goes smoothly. Interactive part is done by VGA interface that can show board as an easy to understand, user friendly visual and important informations such as score and turn to players which they need to decide their next move. Our game logic checks winning conditions and validity of the inputs after every move. Then VGA uptades the board accordingly. This report includes explanations and in depth analysis of various parts and codes from our project. We were able to satisfy nearly every aspect of required project description.

A. Distribution of Tasks:

Our team member Emirhan Yılmaz Güney designed game logic part, by creating the game controller and winning checker modules to implement the rules of the game for a successful playing experience. Mehmet Kurt and Tanel Gülerman implemented the debouncing and VGA sync parts to firstly take the input without any error and synchronize the game outputs with the VGA screen. They were responsible for creating the VGA screen elements and the VGA main module to project the statistics, board and the moves done to the screen.

II. Code Architecture

In the coding part, we utilized four modules, namely debouncer, game controller,VGA synchronizer and VGA main modules, and we used VGA main module as the top module. While debouncer is utilized to take inputs from players, game controller processes the inputs, then outputs the board situation and variables such as move numbers, win numbers etc. and checks the possible win and draw situations. The displaying process is handled by VGA synchronizer and VGA main modules. A diagram of the module structure is shown below in Figure 1.

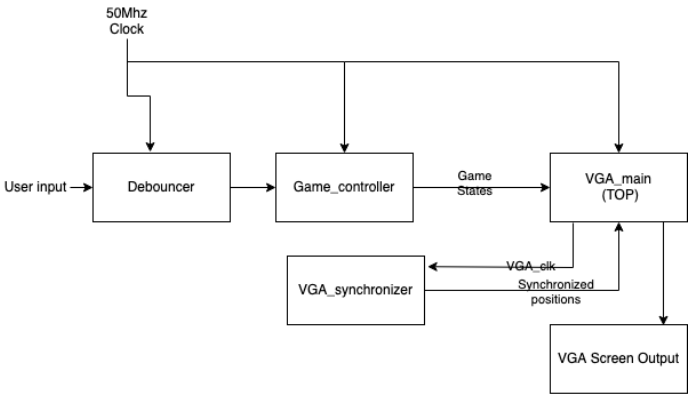

Fig. 1: Module diagram.

The diagram shown in figure is composed of the modules in the code. In the FPGA implementation, there are another connections in addition to the shown ones to the pins and buttons of the FPGA. Only clock connections are shown in the diagram in Figure 1.

III. Pin Connections

Since we used Altera DE1-SoC board, we utilized the pin assignment table of it. The main requirements were to make the connections of internal clock, VGA driving pins and VGA clock. One can see the pin assignment table below (Figure 2).

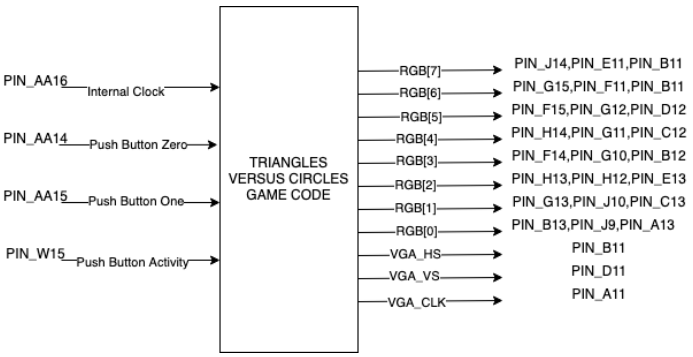

Fig. 2: Pin connections done in Altera DE1-SoC

IV. Getting input and Debouncing the Push Buttons

In our project Triangles versus Circles, we used the push buttons of the FPGA to get input from players. We assigned three push buttons, namely zero, one, and activity button. Since we have a 10x10 board in the game, the inputs should be 4 bits per coordinate and 8 bits in total. As a result, the player will push 8 times in total to the push buttons, 4 times for x coordinate and 4 bits for y coordinate. After pushing the buttons, the last will be pushing the activity button to make the move.

In the ideal case, the working principle of push button is basically a switching mechanism, as shown in Figure 3. However, in real life, this is not the case. Due to non-idealities in the mechanisms, buttons bounce for a while, once they are pushed, as can be seen in Figure 4. When a switch is pressed/released, it sort of shows some 'hesitance' and transits between ON/OFF states for duration of order of micro/milliseconds before it actually settles down to the final stable state [1].

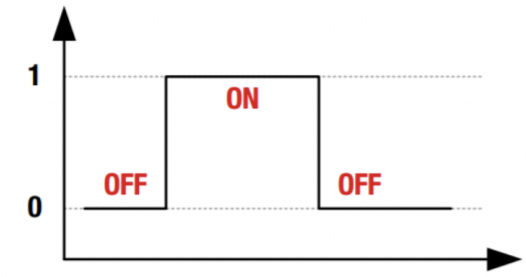

Fig. 3: Ideal switching case.

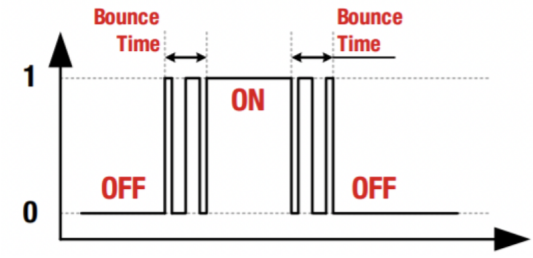

Fig. 4: Non-ideal switching case.

We implemented a debouncing module to solve the issue of bouncing. Firstly, we created counter variables for each buttons, namely zero button, one button and activity button. Then, we update our x-coordinate and y-coordinate variables after 5 clock cycles are passed, accordingly with the button inputs. Thus, the bouncing cycles are ignored by the help of this algorithm. One can see the code pieces of the debouncing of zero button and one button below, in Figure 5.

```
if (button0_counter == 'd5 && button0 == 0 )
        begin
                xy = xy << 1;
                xy = {xy[7:1],1'b0};
        end

if (button1_counter == 'd5 && button1 == 0 )
        begin
                xy = xy << 1;
                xy = {xy[7:1],1'b1};
        end
```

Fig. 5: Code pieces for debouncing.

V. Game Logic / Controller Module

Our game "Triangles versus Circles" has a logic similar to the Tic-Tac-Toe game. Different from Tic-Tac-Toe, we have a 10x10 board and winning condition is placing 4 successive shapes (triangle or circle) orthogonally or diagonally. In addition to this, for each team, the sixth moves clears the first moves and the 12th moves clears the second moves. For instance, when the triangle team done the sixth move, their first move is cleared and gone under a state where no team can place a new shape to there, meaning that the cleared box is anymore a restricted area. In addition to these, the draw condition is the fact that the total move number exceeds 25, meaning that if no team wins the game until the 25th move in total is done, the game goes draw.

```
// horizontal
if (is_won && ((xcoor==resultx && ycoor==resulty+4'b0010) || (xcoor==resultx && ycoor==resulty-4'b0010)) && total_move_num>3)
        begin
                state <= IDLE;
                cir_score <= cir_score + 3'b001;
                player_turn <= 1;
                is_idle <= 1;

                if (ycoor == resulty - 4'b0010)
                begin
```

Fig. 6: Winning statement code piece.

After a win or draw situation, players wait for ten seconds to start the new game while the board is being wiped. In the screen, the move number, win number for each team and player in turn data is shown around the board. This is the whole game logic.

Implementation of the controller in verilog raised difficulties since we didn't utilize SystemVerilog which lets modules take matrix inputs, such as 10x10 board matrix. Therefore, we flattened the board matrix, and hold it in a number instead of a matrix or array. Since each box in the board is represented in 4 states, namely empty, triangle, circle and restricted, we used 2 bits for each box in the board. Thus, we implemented a 200 bit number for the whole board, each 2 bit representing a box.

The game controller consists of three states, namely IDLE, TRIPLAYS and CIRPLAYS. IDLE contains the start, win and draw sessions, whereas TRIPLAYS and CIRPLAYS

defines the triangle and circle user's commands respectively. In the IDLE state, we have a counter to stay in IDLE for 10 seconds, and the rest is about clearing the board and changing the scores. In TRIPLAYS and CIRPLAYS states, the 200 bit board variable is changed accordingly with the given inputs and the move numbers are increased. In addition to these, the checks for if the input is valid or not and the winning and draw conditions are done in these states too, as shown in Figure 6.

For the winning checker, we first have a parameter named iswon, to check if there are three successive orthogonal or diagonal triangles/circles, which is updated in each move by checking the situation of the board. The state diagram is shown in Figure 7. Then, the incoming inputs are checked for their relative positions to these points in order to satisfy winning conditions. This check is done in the TRIPLAYS and CIRPLAYS states if iswon parameter is turned to True by the winning checker part. If there is a winning situation, one game cycle is completed and the algorithm goes back to the IDLE state and the board is wiped.

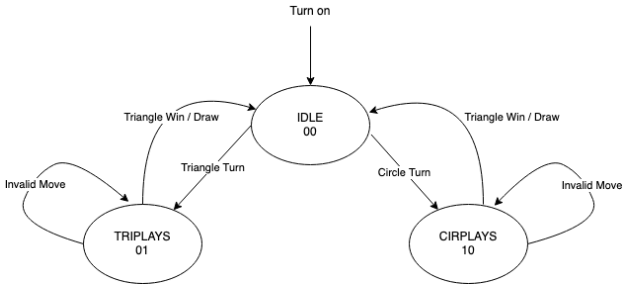

Fig. 7: State diagram for the algorithm.

VI. VGA Implementation

After building the game logic and its semantic architecture concretely, it is necessary to screen it to play the game synchronously. In order to screening operation, we will use VGA interface at a resolution 640x480@60 Hz. VGA stands for "Video Graphics Array" which is commonly used for synchronous video transmission. Protocol of VGA needs five signals; these are the main outputs R(Red), B(Blue), G(Green) channels; HSync and VSync. RGB signals are used for coloring of the each pixel. However, Hsync and Vsync are used for a different task. They are the signals which produce pulse signals at certain times and with certain polarity, to synchronizing RGB signals [2].

Signal	Logic level	Description
VSYNC	5V	Tells the monitor when a screen has been completed
HSYNC	5V	Tells the monitor that a line has been completed
R	0.7V	red color channel
G	0.7V	green color channel
B	0.7V	blue color channel

Fig. 8: Signals of the VGA interface.

Mainly, VGA Implementation had been done in two different piece of code. One synchronozation module is used for directly realize this synchronization via Verilog HDL. Some specific VGA parameters must be given inside a piece of code, such as HPorch and VPorch, as shown in Figure 9.. However, these parameters could be taken as default values which are mainly used in the VGA screening operations.

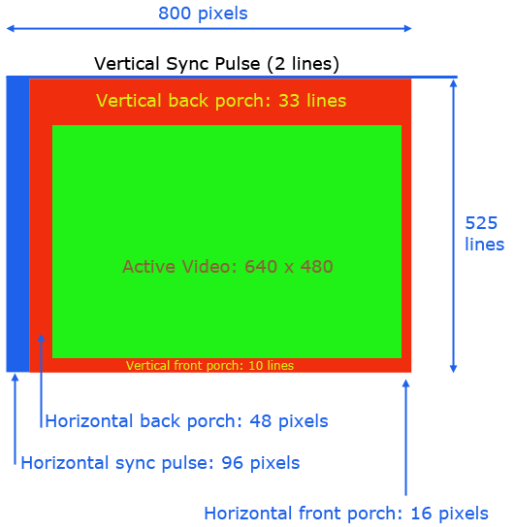

Fig. 9: Hporch and Vporch visualization for VGA implementation [3].

In the game, there should be game board, grid, texts which are used for coordinate system, and circles and triangle images which will be directly controlled and positioned by the player from FPGA push buttons by giving the coordinates in 4 bit binary system.

To create the game board, two main path could be taken. One of them is directly creating the board by drawing from specific programs such as Pixilart or Paint as outputting a 640x480 png or jpg file, then pushing it to FPGA memory. Since some specific transformations applied on the images that we would push to the FPGA, this process will be explained in detail under this section. The other way to creating the board, grids and letters by directly changing RGB values of the range of given coordinates. In the Verilog, we have "posh" and "posv" values which are x and y coordinate values of the screen. After giving the range, RGB values are synchronously printed out to screen in every clock pulse. Moreover, to determine the 10x10 board, we have used some formulation to locate the triangles and circles. In below you could see the piece of code that takes the center point of each cell into a matrix named center coordinates Figure 10.

```
initial begin

        for(k=0;k<10;k=k+1)
                begin
                        for(m=0;m<10;m=m+1)
                                begin
                                        center_coordinates[m][k][0] = 316+k*30;
                                        center_coordinates[m][k][1] = 116+m*30;
                                end
                end
end
```

Fig. 10: Pixel formulation to determine the center point of cells.

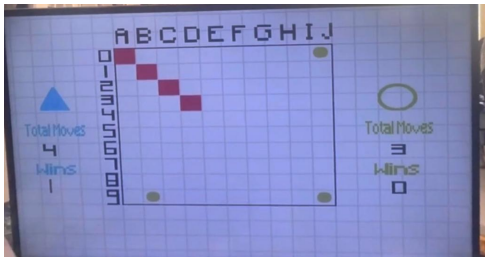

Fig. 11: Example Game Board photo in the project process.

In our project, both of the ways are used to create the game board. To print out the grid lines, triangle and circle shapes, we used the classical way, which is changing the RGB values of given range.

In the above figure (Figure 11), we are giving the specific grid lines coordinate by ”posh” and ”posv” variables. However, since screening all the letters, numbers and game texts by this way is too challenging, we changed our vision into other solid way, which is using FPGA memory. This type of screening an image requires a special process. Firstly, we need to introduce the image to HEX code generator. This piece of code takes the jpeg input, generates an HEX code for each pixel (written to txt file) depending on the size. Then, by using the special keyword ”readmemh” in Verilog, we are reading the HEX codes of each pixel from .txt file. It’s common for a firmware to need data loading into a memory array, ram. Verilog provides for this purpose: ”readmemh” as in Figure 12 [4].

```
$readmemh("winstri.txt", winstri);
$readmemh("winscir.txt", winscir);
$readmemh("alphabet.txt", alphabet);

$readmemh("numbers.txt", numbers);
$readmemh("totalmovescir.txt", totalmovescir);
$readmemh("totalmovestri.txt", totalmovestri);
```

Fig. 12: Example readmemh usage of grid alphabet and numbers.

After reading the HEX codes from txt files for each image that we want to print out to screen, we would determine a specific area of pixels where those HEX lines would match with those pixels. In below figure (Figure 13), there is an example of determining coordinates of the grid alphabet by using posh and posv variables.

```
always @(posedge VGA_clk)
    begin
    // ALPHABET

    else if((63<posv && posv<93) && (304<posh && posh<603)
    && (alphabet[(posh-304)+299*(posv-63)]!=12'b111111111111))
        begin
            r <= {alphabet[(posh-304)+299*(posv-63)][11:8],4'b0000};
            g <= {alphabet[(posh-304)+299*(posv-63)][7:4],4'b0000};
            b <= {alphabet[(posh-304)+299*(posv-63)][3:0],4'b0000};
        end
```

Fig. 13: Example of determining the specific region of pixels

After understanding the process behind the screening by using FPGA memory, we used this approach for print out the basic .png files that indicates the grid alphabet, grid numbers, text of total moves, text of draw and etc. Moreover, since game table requires dynamic changes such as total number of moves, total number of wins numbers, we have uploaded photos of each numbers. By using some if conditions, numbers are screened in the game table in a easy way.
To conclude the VGA, there are consisting of 2 main codes, the sync module, which is the first main part of this implementation, provides continuous updating of the 640x480 screen projected to VGA. There are Vsync and Hsync which provide Vertical and Horizontal syncs along with VGA’s RGB outputs. The other module, VGA main module provides the print outs such as grid numbers, alphabets and texts by using the HEX matching.

VII. Discussion and Summary

To conclude we have designed an interactive game using Verilog HDL and FPGA boards given to us by department. In this project we have learned various things ranging from game logic controller coding to VGA interface implementations. This project was really rich in terms of design choices and number of challenging situations that one must have to go through. Our team learned a great deal about Verilog HDL designing and coding in general while working this seemingly simple yet uniquely interesting project in the sense that one must control every aspect of the project from scratch.

References

[1] Chipmunk, *Debouncing switches in verilog / vhdl – chipmunk logic*, Dec. 2022. [Online]. Available: https://chipmunklogic.com/digital-logic-design/debouncing-switches-in-verilog-vhdl/.

[2] G. Pacchiella, *Implementing vga interface with verilog*, Jan. 2018. [Online]. Available: https://ktln2.org/2018/01/23/implementing-vga-in-verilog/.

[3] N. Dumont, Jul. 2011. [Online]. Available: https://nathandumont.com/blog/vga-primer.

[4] W. Green, *Initialize memory in verilog*, Jul. 2021. [Online]. Available: https://projectf.io/posts/initialize-memory-in-verilog/.